構文誤りを含むプログラムのブロック言語表現への変換

構文誤りを含むプログラムのブロック言語表現への変換
Translation from text program with syntax errors to block-based representaion

山梨　裕矢
Yuya Yamanashi
法政大学大学院 情報科学研究科
E-mail: 17t0024@cis.k.hosei.ac.jp

Abstract

Block-based visual languages are often used by novice programmers. Some blocks environments provide a bi-directional transformation between a traditional text language and a block-based language. Such dual-mode tools provide that users may benefit from the learnability of blocks in block-based mode, while they learn syntax and get the efficiency of text in text-based mode. However, incomplete code in block-based language may produce syntactically erroneous program. Such incomplete code cannot be parsed, so they aren't converted back to blocks using normal parsers. In this research, we focus on the characteristics of syntax errors in program generated from incomplete code in block-based language and propose a method to predict correct syntax for incomplete code. A PEG (Parsing Expression Grammar)-based parser enables to parse incomplete program and predict correct syntax based on keywords necessarily generated from each block. Then, phantom tokens are inserted for incomplete part of code to obtain parsable code. This also enables to convert syntactically incomplete program text to visual blocks. Using this method, we implemented a bi-directional converter between JavaScript language and a block-based language created with Blockly. Even programs containing syntax errors in JavaScript caused by generating from incomplete block representation can be translated back into the original block-based program. In addition, it enables immediate conversion from a partial code fragment typed by a programmer to blocks.

1. はじめに

2020 年にプログラミング教育が必修化されるなどプログラミング学習の需要が高まっている．プログラミング初学者向けの学習環境としてブロック型のビジュアルプログラミング言語が用いられることがあるが，これはプログラムの命令をブロックで表現したもので，ブロックの組み合わせによりプログラミングを可能にするものである．ブロック型言語だけでなく一般的なテキスト記述形式のプログラミングも学べるように二つの表現を組み合わせ両表現間で双方向に変換できるプログラミング学習環境も存在する．しかし，この変換は構文誤りがない時に行われるもので，コードの一部に誤りがある場合テキストからブロックへの変換は行われない．ブロックの組み合わせでプログラムを表現するため，他のブロックと組み合わせる前や値を入力する入力部が空白の間は構文誤りを起こしてしまう．このようにブロック型言語でプログラムを組む場合には構文誤りが頻出し，その度ブロックへの変換が行えず二つの表現を組み合わせる利点が得られにくくなる問題点が挙げられている [1].

本研究では「欠けた」ブロックから出力されたコードからでも再度ブロックに変換できるように拡大文法に適用し修正することで構文誤りを取り除くようにした．各ブロックに元の文や式を識別できるキーワードを必ず出力するようにしておく．PEG [2]で定義した文法を用いてキーワードを手がかりに構文の推定を行い，構文で不足している箇所については仮のトークンを置くことでブロックへの変換時には構文誤りが起きないようにした．

この手法を JavaScript 言語と Blockly で作成したブロック型言語の変換に対して実装を行い，実験として各ブロックの一部を「欠けさせて」いったときにその出力されるコードから元のブロックに戻るか，また，テキストを編集した場合にどのような誤りではブロックに変換できないか調べた．さらに，拡大文法の適用から構文誤りの修正までにかかる時間を計測しどの規模のプログラムにまで適用できるかについて調べた．

2. 関連研究

2.1. Tiled Grace [3]

Tiled Grace は Grace という教育向けテキスト型言語とそれに対応するブロック型言語の両方を備えたプログラミング環境である．ブロック型言語による学習からテキスト型言語への学習移行を目的に作られており，アニメーションで両表現を変換させることによりテキストとブロックの対応づけを示している．しかし，欠けているブロックが存在する場合，テキストに変換することは出来ず，テキスト側に構文誤りが存在する場合にもブロックへ変換することができず両プログラミング表現の併用が構文誤りによって中断されてしまう．

2.2. Pencil Code [4]

Pencil Code はブロック型言語を用いながら汎用的なテキスト型言語である JavaScript と CoffeeScript を学習できるようにしたプログラミング環境である．ブロックエデ

Supervisor: Prof. Akira Sasaki

ィタとテキストエディタの二つを備えており，クリックひとつでそれらを切り換えることができる．ブロックは値や式を入力するテキストフィールドを持っているが空白であった場合はブロックの空白情報をテキストでは__(アンダーバー二つ)で表現しブロックが不完全な状態でもテキストに変換できるようになっている．しかし，テキストからブロックに変換する際は構文誤りのない形でないと変換することができない．

3. 提案手法の概要

ブロック型言語とテキスト型言語を併用する場合，ブロックからテキストへの変換は常に可能であるが，テキストからブロックへの変換を行う際にはプログラムに構文誤りがない状態でなければ変換できない．これはブロック型言語への変換時に抽象構文木が必要とされるため，構文解析を行う必要があるからである(図 1).

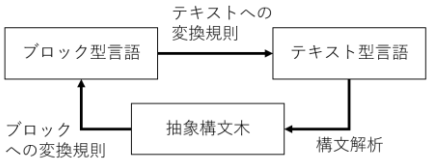

図 1：ブロック型言語とテキスト型言語の変換

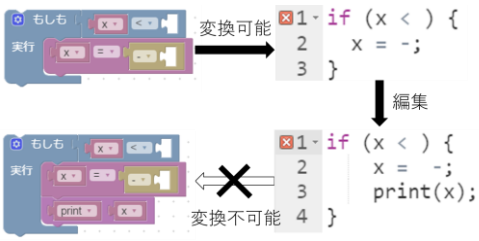

図 2：欠けたブロックと構文誤りの発生

ブロック型言語において，ブロックの組み合わせでプログラムを組む性質上，図 2 左上のような「欠けた」ブロックが発生する事が多々ある．この時，テキスト型言語への変換は可能だが構文解析が行えないため，構文誤りを取り除く以外にテキスト側に変更を加えてもブロックにその変更は反映されない．本研究では，テキストプログラムに構文誤りが発生した状況でも，その処理ができるように対象となるテキスト型言語の文法よりも拡大した文法を定義し誤りを修正するようにした．このような文法を拡大文法と呼ぶ．拡大文法は，各構文におけるユニークなトークン列を「キーワード」とし，それ以外のトークンを省略可能にした構文を用意し，これをもとの構文に統合することで構成される．

このようにして，構文誤りがある場合にも，拡大文法を用いて，解析，変換できるようになればテキストからブロックへの変換の即時性が増す．たとえば，if 文をブロックに変換する場合，JavaScript 言語の場合は *if (式) { 文 }* の形のテキストでなければ，構文解析が出来ずブロックに変換することが出来ないが，本手法を用いれば *if x <* のように構文の途中の段階でもブロックへ変換可能になり，変換の即時性が増す．

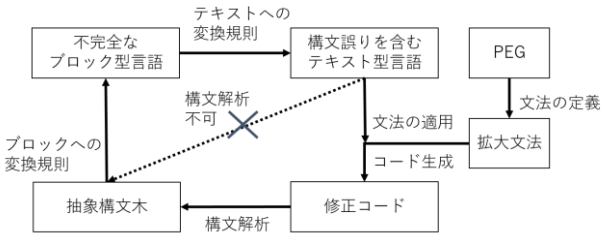

図 3：構文誤りが含まれる場合の変換

構文誤りが含まれる場合にテキスト型言語からブロック型言語に変換する方法を図 3 を用いて概説する．「欠けている」ブロックから出力されたコードに対し，拡大文法の適用を行い，キーワードから構文の推定をし，構文で不足している箇所に対して補完を行うことで構文誤りのないコードに変換する．この修正後のコードを構文解析し，得られた抽象構文木からブロック表現に再度変換しなおす．テキスト表現に変更を加え，構文誤りが発生した場合にも同様の手順でブロック表現への変換を行うが，ブロックから出力されたコードとは異なり，拡大文法にも則さないコードが与えられる可能性がある．この場合にはブロックへ変換することはできない．

4. 拡大文法の作成とブロックへの変換法

3 節で述べたように，本研究では，構文誤りに対応できるようにするため文法の拡張を行うアプローチをとる．本節では，拡張文法の作成法と，その文法をもとにして，構文誤りを含んだテキストをブロック表現へ変換する方法について説明する．

4.1. 誤りの種類と対処する方針の概要

多くのプログラミング言語において，数値や真偽値のようなリテラルや変数名などの単体のトークンで構成されているものを除き，各文・式は複数のトークンから成る文法になっている．したがって通常，各文や式に対応するブロックを用いた式は部分的に「欠けた」状態でも，構文に特徴的なキーワード等を残した状態のコードを出力させることができる．このように「欠けた」ブロックから出力されるそのコード断片から各文・式を識別するキーワードを手がかりに構文の推定を行う．このようなキーワードは 1)ブロックが欠けた状態でも必ず生成される単語であり，2)他の文・式に設定したキーワードと一致しないものを選ぶことが必要になる．構文推定をした後，その構文で不足している箇所に対しては，記号や仮のトークンを置くことで構文誤りを回避する．具体的には，不足しているものが*( )*や*[ ]*, *;*などの区切り文字であった場合はそれを補完し，文や式であればそれに対応する仮トークンを置くことで構文解析可能なコードにする．

一方で，テキスト型言語の編集によって起きた構文誤りの場合には，変換対象言語の文法から逸脱したもの，つまり拡大文法にも当てはまらない場合がある．この時，誤りの箇所が特定できる場合はその部分をアノテーション付きの文字列にして，エラーブロックに変換する方法をとる．誤りの範囲が正確に特定できない場合，もしく

は誤りだと思われる箇所を文字列に置いても構文誤りがなくならない場合はブロックに変換することはできない.

4.2. 文法の拡大方法

「欠けた」プログラムの解析が可能となるように，正しい構文に「欠けた」コードを扱うための構文を追加する．これは，まず，各構文の中でキーワードとなるトークン以外を省略可能な構文要素として指定することで，可能となる．このようにすることで，キーワードの並びのみで構文を解釈可能となり，もとの構文を推定できる.

```
/* E, S は非終端記号で E は式，S は文,
   U はキーワード t1, t2 は終端記号 */
A = E U E t1 S t2
/* 拡大後 */
A = E? U E? t1? S? t2?
/* E は非終端記号，U'はキーワード t は終端記号*/
B = E (U' E)*
/* 拡大後(失敗例) 空文字を受理してしまう*/
B = E? (U' E?)*
/* 拡大後(成功例). __は空白文字列 */
B = E (U' E?)*
/ __? U' B*
```

図 4：拡大文法の定義

たとえば，PEG で定義した文法 A が図 4 のような構文であった場合，キーワードに設定しているトークン U 以外は省略可能にする．ここで ? は PEG の文法で省略可能，つまり parsing expression が 0 個か 1 個存在する場合に受理することを意味する．図のように変更することで構文に U が存在する時に A であると推定することができる.

ここで，構文の中に繰り返しがある場合は空文字を受け付けないような文法に変更する必要がある．PEG 上で B が上記のような繰り返しを持つ構文で，キーワード以外のトークンをすべて省略可能にすると，*が 0 個以上の繰り返しを表すので，空文字を受け付けてしまう文法になり，PEG パーサの性質上このような文法を定義することは出来ない．そこで構文のはじめの parsing expression が存在する場合としない場合に分け，どちらの場合でも空文字を受け付けないような構文を表現できる.

4.3. 拡大文法によるコード生成法

拡大文法で定義されている構文規則にマッチした時に同時に誤りを訂正したコードを出力する．方針は 4.1.項で述べた通りである.

```
A = e1:E? u:U e2:E? t1? s1:S? t2?{
  e1 = e1 ? e1 : putPhantomExpr();
  e2 = e2 ? e2 : putPhantomExpr();
  s1 = s1 ? s1 : putPhantomStm();
  return e1 + u + e2 + t1 + s1 + t2;
}
```

図 5：拡大文法に対するコード生成定義

省略可能なトークンにおいて，トークンが存在しない場合 null が値として入ってくるため図 5 のように三項演算などを用い，仮のトークンを置く．t1, t2 のような終端記号の場合はその有無に関わらず挿入することで補完する．仮トークンは構文として不足している箇所の種類が式か文かに応じて変える．式を置く場合には普段のコードではあまり使用されないようなものが，文を置く場合には空文や空のブロック文などが望ましい.

4.4. ブロック表現への変換方法

テキスト表現からブロック表現へ変換するアルゴリズムを擬似コードによってあらわしたものを図 6 に示す.

```
function codeToBlock(code, isFirst) {
  if(isFirst == false) return error;
  if (errorExists(code)) {
    fixedCode = fix(code);
    if(codeToBlock(fixedCode, false) == error)
     return error;
  } else
    ast = parseScript(code);
  for(statement of ast.body){
    block = statementToBlock(statement);
    deploy(block);
  }
}
function statementToBlock(statement) {
  block = createBlock(statement.type);
  for(property of statement.property){
    if(isPhantomToken(property))
      continue;
    else if(isExpression(property))
      combineBlock(block,
        expressionToBlock(property));
    else if(isStatement(property))
      combineBlock(block,
         statementToBlock(property));
    else
      changeBlock(block, property);
  }
  return block;
}
```

図 6：抽象構文木からブロックへの変換擬似コード

codeToBlock 関数は変換元のテキスト表現である code とこの関数が初めて呼ばれたかを表すフラグ isFirst を受け取り，ブロックへ変換した後そのブロックをエディタに配置する関数である．code に構文誤りが存在する場合は拡大構文の適用を行い，修正後のコードを引数に再帰的に codeToBlock 関数を呼び出すが，拡大文法に則さないような構文誤りなどがある場合に抜け出せるよう，フラグを false にする．誤りがなくなった後，構文解析を行い抽象構文木の情報を得る．構文木の根の直下にある文の列に対してそれぞれブロックへ変換するが，その文の構文木情報を引数に statementToBlock 関数を呼ぶ．statementToBlock 関数では引数として渡された構文木である statement から文の種類に対応するブロックを createBlock 関数で作成し，構文に付随する情報が文や式であれば再帰的に statementToBlock 関数もしくは expressionToBlock 関数を呼びブロックを取得してくる．そこで得られたブロックと初めに作ったブロックを結合し一つのブロックを作っていく．もし構文に付随する情報が終端記号であった場合はその情報により初めのブロ

ックの内容を変える．仮トークンであった場合はブロックへの変換をせず，ブロックが「欠けた」状態にする．

5. 実装

本手法をテキスト型言語である JavaScript 言語と Blockly[5]で作成したブロック型言語の変換に対して行った．

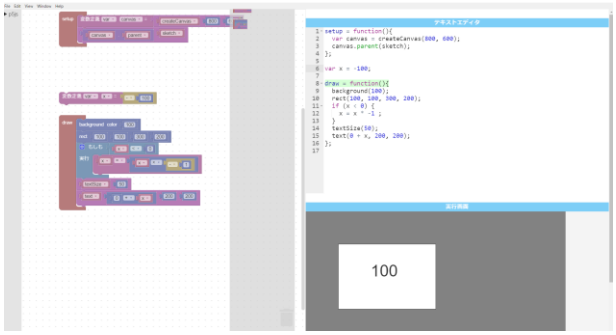

図 7：操作画面

図 7 は実装を行ったプログラミング環境の操作画面である．左部はブロック型言語を用いたエディタとなっており，ブロックの操作を行うと右上部のテキストエディタ内容もそれに応じて変わる．テキストエディタで編集を行った場合はタイピングが止まってから 1 秒後にブロックに変更されるようになっている．右下部は実行画面となっており，プログラムの実行結果が表示される．

ブロックエディタ上でダブルクリックをすると新たなブロックエディタとテキストフィールドが現れ，その入力部にプログラムを打ち込むとブロックへ即時変換される．テキストエディタと違い比較的短いコードの変換を行うためブロックの即時変換が可能になっている．ブロックの検索や，*1+2*n* のようなテキスト型言語で組むには容易だがブロック型言語で作るにはコストのかかるようなプログラムの作成用にこのエディタ部設けている．

5.1. ブロック型言語の設計と実装

ブロックを作る際にはその構文のキーワードとなる単語を必ず出力するようなブロックにしなければならない．たとえば，比較演算のブロックにおいて左右の式になる箇所のブロックの付替えは可能にしても良いが，キーワードとなる比較演算子を別ブロックとして分離し付替え可能にしてはいけない．

ブロック型言語の作成には Blockly を使用した．Blockly の JavaScript 用サンプルで用意されているブロック以外にも自作し，ArrowFunction や ClassDeclaration などほかの構文で代替する事が可能な構文を除くほぼすべての JavaScript 文法をブロックで使用できるようにした．

5.2. 構文解析とブロック表現への変換

構文解析には Esprima を使用する．Esprima は JavaScript 向けパーサであり，Mozilla が提唱する Parser API に準拠した抽象構文木を得ることができる．この抽象構文木は根に Program というノードを持ち，その直下に Statement や Declaration のリストを持つ．プログラム全体をブロックに変換する際はこの Statement と Declaration の一つ一つをブロックに変換する．

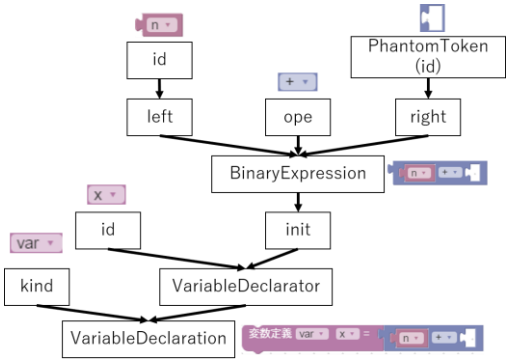

図 8：VariableDeclaration の構文木とブロックへの変換

図 8 は var x = n + のコードを拡大文法に適用し修正したコード var x = n + _;の抽象構文木からブロックへの変換を行った例を示した図である．構文木に対し後順走査を行い，葉のノードからその構文に対応するブロックを生成して親のノードへとブロックを組み合わせていく．今回は仮トークンに__(アンダーバー)を設定しているが JavaScript の文法では変数名として識別されため，そのままブロック型言語へ変換すると名前が_の変数ブロックへ変換される．しかし，空白箇所を仮トークンとしておいているためここでは空白のあるブロックが出力されるべきである．そこで，ブロック型言語への変換時に仮トークンはブロックへ変換しないことにする．

5.3. 設定するキーワード

表 1 に今回 JavaScript の式や文に設定したキーワードの一例を示した．表に挙げたものは特に注意するべきキーワードで，ほかのキーワードと衝突する可能性のあるものである．

表 1：JavaScript の各構文に設定したキーワード例

VariableDeclaration	var / let / const
AssignmentExpression	= / += / *= など
BinaryExpression	Expr (+ / - / * / == など)
UnaryExpression	+ / - / ! など
ForStatement	for
For-InStatement	for in

たとえば，VariableDeclaration で＝をキーワードに設定してしまうと AssignmentExpression と重なってしまうため，変数定義の種類をキーワードにしている．BinaryExpression と UnaryExpression ではどちらも＋やーなどの演算子が出てくるため，これらの記号の前に UnaryExpression では式が来ることはない事に注目し BinaryExpression のキーワードを式と算術演算子の列に設定した．for 文のような様々な書き方がある文に対してはそれらに共通する単語，ここでは for をキーワードにし，そのキーワードに続く式や記号から構文を推定していく形にした．

5.4. PEG.js [6]による拡大文法定義

本システムでは PEG.js のサンプルに含まれる JavaScript の文法をもとに，構文のキーワードとなる一部から構文を推定し足りない箇所を補ったコードを出力するものにした．PEG.js は JavaScript 向け parser generator で文法定義とその文法に対するコードの生成を行うことができる．非終端記号の場合は=の右辺に生成規則を記述する．右辺の終端記号/非終端記号には名前をつけることができ，これをアクション部である{}ブロック内の JavaScript コードから参照することができる．

```
IfStatement =
IfToken __ "("? __ test:Expression? __
")"? __ cons:Statement? {
   e = test ? test : putPhantomExpr();
   s = cons ? cons : putPhantomStm();
   return "if(" + e + ")" + s;
}
```

図 9：if 文に対する拡大文法

図 9 は if 文の推定を行う拡大文法の例である．if 文で設定されているキーワードは *if* でそれ以外の構成要素である括弧や条件式，は省略可能にした．括弧に関しては足りない部分に対して括弧の挿入を行い，条件式に対しては仮のトークン(ここでは_を仮のトークンとしている)を挿入し構文誤りを回避する．

6. 実験

実験では「欠けた」ブロックから出力されたコードから元のブロックに変換できるか，また，打ち込んだテキストからブロックに変換できるかを確認した．また，構文誤りのあるコードを拡大文法に適用しコードを修正するのにかかる時間について調べた．

6.1. 誤りを含むコードからブロック表現への変換実験

「欠けた」ブロックから出力された誤りを含むコードから元のブロックに再度変換できるか調べた．実験手順としては 1) ブロックの生成を行い，2) 他ブロックとの接続箇所や入力フィールドを空にしていき，3) 出力されたコードからブロックへの変換を行って 4) 2)のブロックと同じブロックが再度生成されるかを確認した．

この時，今回拡大した JavaScript 拡大文法 32 構文中 29 構文は再び同じブロックに変換することができた．以下には元のブロックに戻らなかった構文を示す．

(a) BinaryExpression において演算子が＋，－の場合，演算子の前の式を取り外すと，UnaryExpression に変化してしまう．たとえば n + 2 のような式があった時に n のブロックを外すと+ 2 に変わり，UnaryExpression と解釈されてしまうようになる．

(b) UpdateExpression において式を取り外すと後置演算子なのか前置演算子なのか判別できなくなる．たとえば，++i という式で i を外すと++になり，この演算子が前置のものなのか後置のものなのか判断できなくなる．

(c) ForStatement では初期化の箇所に Expression が来る場合と VariableDeclaration が来る場合があり二種類のブロックを用意している．VariableDeclaration のブロックの場合に初期化文を取り外すと Expression の ForStatement になってしまった．

6.2. テキスト編集による誤りからブロック表現への変換が出来なかった例

テキストを編集しブロックに変換させていく中でブロックに変換できない，もしくは意図しないブロックに変換される例を挙げていく．

ブロックから出力されたコードとは異なり，テキストからブロックへ変換する際には拡大した文法にも則さないコードが与えられる可能性がある．

```
/*(1)構文の構成要素が間違っている場合 */
if(x < 0;){ }
/*(2)括弧の対応が取れていない場合 */
if(x < 0){
    y = 1;
z = 100;
function f(){ }
```

図 10：テキスト表現からブロック表現への変換失敗例

図 10 の(1)のようにキーワードが含まれている場合でも構文として正しくない要素が入っている場合には変換する事ができない．*if* の後の括弧内には式が来なければいけないが，例では ExpressionStatement が来るため文法にマッチせず if 文と推定することができない．しかし，コードが if (x < 0; や if (x < 0; { }の場合は変換が可能になる．前者の場合は x < 0 が Expression にマッチし; が EmptyStatement にマッチするためで，後者の場合は x < 0 が Expression に，;が EmptyStatement として Statement にマッチし{ }は BlockStatement として IfStatement の一部としてではなく別の Statement としてマッチする．このようにその構成要素の位置が特定でき，かつ文法として期待される構文の種類とは全く異なるものでない場合のみ変換ができない．

(2)の例は文法としてマッチするが，変換が意図したものではない可能性がある例である．条件式の後の{ }対応が取れていないため BlockStatement として解釈され Statement とマッチする限り先を読んでいく．最後の行で FunctionDeclaration が来るため z = 100;のあとに BlockStatement の閉じ括弧を挿入する．このように文字列の""や括弧などの開始と終了が組になっている構文の終了が存在しない場合，意図した箇所に終了が置かれるとは限らない．

6.3. 変換時間

初めに一種類の Statement に対してその個数を 1 個の場合と 100 個の場合で，変換にかかる時間がどれくらいかかるのか調べた．比較のためにそれぞれ誤りがない場合についての変換時間も記載している．

表 2 をみると a=0;の ExpressionStatement と if(a){}の IfStatement，try{}の TryStatement の順に変換時間がかかっているが，これは Statement における探索順序が後のほど探索に時間がかかるためである．また，代入式に関して左辺を削除した場合と右辺を削除した場合では前者の

方が変換に時間がかかるのは左辺で探索する範囲よりも右辺で探索する範囲が広いためである．

表 2：誤り構文が一種類の場合の修正時間

繰り返し回数	a = 0;	= 0;	a = ;	= ;
1 回(ms)	4.35	4.46	4.98	4.95
100 回(ms)	19.6	17.6	137	134
繰り返し回数	**if(a){}**	**if(){}**	**try{} catch(e){}**	**try{} catch(){}**
1 回(ms)	8.06	13.4	9.88	15.0
100 回(ms)	340	757	459	912

次に，ある JavaScript のプログラム(162 行)に対してコード修正を行うのにかかった時間を調べた．誤り箇所はランダムに選択したがブロックが欠けた時に起こるような誤りにしている．コード修正を行うためにはどの文法にマッチするのか探索するのに時間がかかるため，誤りなしでも各構文とのマッチに時間がかかる．そのため，誤りがなかった場合との実行時間の差をとり誤り修正にかかる時間を計測した．

表 3：誤り構文が複数種類存在する場合の修正時間

誤り数	実行時間(ms)	誤りなしとの差(ms)	1 箇所あたりの平均(ms)
0 箇所	2372	0	0
50 箇所	2835	463	9.26
100 箇所	3692	1320	13.2
150 箇所	4101	1729	11.5

表 3 の 1 箇所あたりの平均変換時間を見ると，誤り箇所をランダムに発生させたためばらつきがあるがおおよそ 10ms の時間がかかっているのが分かる．

7. 考察

本研究では「欠けた」ブロックから出力される誤りを含むコードから再び元のブロックに変換できるようにするのが一つの目標であった．6.1.項で挙げた構文以外はすべて変換することができ，欠けたブロックが存在する状態でもブロック型言語とテキスト型言語の併用を止めることなく行うことができるようになった．二つの例で元のブロックに変更することができなかったが，対処法としてはブロックが空いている箇所に対して仮トークンを出力させテキストエディタ上のコードに直接埋めてしまう方法が考えられる．誤りを含むコードの修正を行う際には仮トークンを置くのは内部的なものであり，ユーザ側には表示させないようにしていたが，この場合はテキストエディタ上に直接仮トークンを埋めてしまう．ここで仮トークンを__だとして先の例に対してこの操作を行えば，出力されるコードは__＋2 となり「欠けた」ブロックに再度変換することが可能となる．しかし，テキスト側のコードが本来は＋2 であるところが，仮のトークンが挿入されてしまい厳密には意味が変わってしまう．

テキストからブロックへの変換を行うときには 6.2.項で述べたような場合にブロックへの変換が出来なかった．構成要素が間違っている場合の対処法としては，文法の拡大時に必要な要素を省略可能にするだけでなく，不要な構成要素ともマッチさせ，不要な構成要素があった際には読み飛ばしたりエラーコードブロックとして解釈させるなどの処理をする方法が考えられる．しかし，これを行うと文法の探索数が増えすぎてしまい，コードの修正にかかる時間が増えてしまう．また，ブロックに変換できる場合においてもテキスト表現にある空白情報や算術演算における括弧情報などの抽象構文木で消失してしまう情報を保持できない問題もあった．双方向変換技術を用いて抽象構文木と具象構文木の間の変換を行うことで情報の消失をなくせるのではないかと考える．

実行時間に関して，式や文の変換に関して一つあたり 10ms 前後であったため，5 節で説明した 1 文ごとのテキストからブロックへの即時変換機能に拡大文法の適用によるコード修正を用いるのは現実的だとわかった．今回 JavaScript 言語に対する誤り修正を行ったが，JavaScript 上で誤りの位置を特定するのが難しく，エラーが存在した場合全体にコード修正を行わなければならなかった．大きなプログラムに対応するためにはエラーの箇所を特定し部分適用する必要がある．

8. おわりに

本研究では誤りのあるコードを拡大文法に適用し修正することで構文誤りがある状態においてもブロック型言語とテキスト型言語の間の変換を可能にさせた．キーワードが衝突し，複数の構文として考えられる場合においても仮のトークンや適当な式を置くことによって，テキスト側のプログラムに多少の違いが生じるものの元のブロックへの変換を保証することができる．

今後の課題として，双方向変換の技術を取り入れることにより，キーワードが衝突した場合にテキスト側も変えず元のブロックに変換ができるようにしたりテキスト側の空白情報や括弧などの抽象構文木で消失してしまう情報などを保持しながら変換を行えるようにするほか，テキスト側の誤り箇所の特定を行い誤り修正の部分適用により短時間の変換を可能にすることが望まれる．

文　献

[1] BAU, David, et al. Learnable programming: blocks and beyond. Communications of the ACM, 2017, 60.6: 72-80.

[2] FORD, Bryan. Parsing expression grammars: a recognition-based syntactic foundation. In: ACM SIGPLAN Notices. ACM, 2004. p. 111-122.

[3] HOMER, Michael; NOBLE, James. Combining tiled and textual views of code. In: 2014 Second IEEE Working Conference on Software Visualization (VISSOFT). IEEE, 2014. p. 1-10.

[4] BAU, David, et al. Pencil code: block code for a text world. In: Proceedings of the 14th International Conference on Interaction Design and Children. ACM, 2015. p. 445-448.

[5] FRASER, N., et al. Blockly: A visual programming editor. URL: https://code. google. com/p/blockly, 2013.

[6] MAJDA, David. PEG. js: Parser Generator for JavaScript. URL: http://pegjs. majda. cz, 2011.